Freie Universität Berlin

Volker Roth

Rechnersicherheit, SoSe 21

Übung 03

TutorIn: Oliver Wiese
Tutorium 02
Materialien: Latex, VSC, Skript

26. Mai 2021

---

1 Shoulder surfer meets Markov

(a) *Give an algorithm with observed digits (less than 4) as input and outputs three likely PINs.*

1. We assume that we always observe only the first digit of the PIN.
2. First we preprocessed the Markov model that we expect to get thought a file. We expect this form of representation, because we defined it like that in the last exercise sheet. The following snipped shows the form we expect of a Markov model:

```
    "00": 0.09766187759656811,
    "01": 0.10964242399751109,
    "02": 0.09521532801048535,
    "03": 0.08118016683277238,
    "04": 0.07375969673496395,
    "05": 0.08015285944438737,
    "06": 0.0798709585070059,
    "07": 0.07797165702734245,
    "08": 0.09059390538937141,
    "09": 0.08103410417091668,
    "0\u22a4": 0.11190444775411616,
    "10": 0.13491879837536758,
```

Here is the Markov model after we preprocessed it. Now it is ordered from highest possibility that that char is taken to lowest:

```
    "0": [
        [
            0.10964242399751109,
            "1"
        ],
        [
            0.09766187759656811,
            "0"
        ],
        [
            0.09521532801048535,
            "2"
```

And here is the code on how we did it:

```
#returns a better dict format of markov_model
#following will be returned:
#eg: '0' : [('5',p1),('2',p2),('9',p3)...]
#    '1' : [('2',p1),('6',p2),('0',p3)...]
# where p1>p2>p3. so the possibilitys are sorted.
#that will allow us to get the 'best' edge if we search for
# an edge for a specific char (eg: result['1'][0][0] will return '2')
def preprocessing(markov_model):
    #edge from v->e
    #note that the edges of list_markov are sorted.
    #eg '00':p, '01':p, '02':p, '03':p
    # where p stands for a possibility
    list_markov = list(markov_model)
    v = list_markov[0][0]
    temp = []
    result = {}
    for edge in list_markov:
        #if a new different v1 (char) occurs
        #we can save our current temp in the result
        #and start to calc for the new v1(char)
        #because the previous v will not occur again
        #as list_markov edges are sorted
        if v != edge[0]:
            result[v] = sorted(temp, reverse=True)
            v = edge[0]
            temp = []

        #append the possibility and e
        #but only if e != top
        #as our pin has a fixed len, we dont the end symbol
        if edge[1:] != top:
            temp.append( ( markov_model[edge], edge[1:]) )

    #ad the last digit (9) will be skipped otherwise
    result[v] = sorted(temp, reverse=True)
    return result
```

3. After that we ran the following function. It takes the Markov model, the processed Markov model, the observed digit, the number of PINs we want to get and an accuracy int as input. The function is greedy and always takes the next digit with the highest possibility, which is quite easy, because of the preprocessing we did with the Markov model. After it found more PINs than 'num_of_output_pins' it drops the PIN with the lowest total possibility.

```
def guess_password(ordered_markov_model, markov_model ,accuracy, observed_digit,
    num_of_output_pins):
    result = []
    for i_1 in range(accuracy):
        #add the first (second) char to the pin and calc the probability
        temp_prob_1, temp_char_1 = ordered_markov_model[observed_digit][i_1]
        pin_1 = observed_digit + temp_char_1
        prob_1 = 1 * temp_prob_1

        for i_2 in range(accuracy):
            #add
            temp_prob_2, temp_char_2 = ordered_markov_model[temp_char_1][i_2]
            pin_2 = pin_1 + temp_char_2
            prob_2 = prob_1 * temp_prob_2

            for i_3 in range(accuracy):
                #add
                temp_prob_3, temp_char_3 = ordered_markov_model[temp_char_2][i_3]
                pin_3 = pin_2 + temp_char_3
                prob_3 = prob_2 * temp_prob_3

                #last char (digit) of the pin has to be the top symbol top = '\u22A4'
                #so add that to the pin and calculate the probability of that edge
    with the normal markov_model
                pin_3 += top
                prob_3 *= markov_model[temp_char_3+top]

                #append result and drop lowest
                result.append( (prob_3, pin_3) )
                drop_lowest(result, num_of_output_pins)

    return result
```

(b) *Discuss the efficiency and success rate of your algorithm.*

- The efficiency and success rate of your algorithm dependents on the accuracy variable. As our algorithm has three for loops, its efficiency is accuracy^3.
- Accuracy should/ can not be higher than the length of the alphabet, that is 10. So if we choose accuracy = 10 we basically try each possible combination.
- However with accuracy = 10 our algorithm always succeeds and returns the most likley 3 PINs. When we lower the accuracy we might not get the best 3 PINs, but in our tests half of that (5) seemed sufficient. What follows is a picture of the 3 most likely PINs of the Markov model from the RockYou dataset and how different accuracy values affect these PINs.

```
>>> main(2)
[(0.003192080901017929, '1234T'), (0.0008987235750735024, '1230T'), (0.0008493872205011611, '1212T')]
>>> main(4)
[(0.003192080901017929, '1234T'), (0.0009438533961205413, '1223T'), (0.0013149627524600077, '1123T')]
>>> main(6)
[(0.003192080901017929, '1234T'), (0.0025328192843752473, '1256T'), (0.002250275131115612, '1456T')]
>>> main(8)
[(0.003192080901017929, '1234T'), (0.0025328192843752464, '1256T'), (0.0022502751311156075, '1456T')]
>>> main(10)
[(0.003192080901017929, '1234T'), (0.002532819284375247, '1256T'), (0.0022502751311156066, '1456T')]
```

(c) *Revisit the first-order Markov model from the previous assignment. What are your guesses if the adversary observes only a one.*

- The expected input Markov model (snipped):

```
    "00": 0,
    "01": 0.16666666666666666,
    "02": 0,
    "03": 0.5,
    "0\u22A4": 0.3333333333333333,
    "10": 0.1111111111111111,
    "11": 0,
    "12": 0.2222222222222222,
    "13": 0.4444444444444444,
    "1\u22A4": 0.2222222222222222,
    "20": 0.2222222222222222,
    "21": 0,
    "22": 0,
```

- After the preprocessing (snipped):

```
    "0": [
        [
            0.5,
            "3"
        ],
        [
            0.16666666666666666,
            "1"
        ],
        [
            0,
            "\u22a5"
        ],
        [
            0,
            "2"
        ],
        [
            0,
            "0"
        ]
    ],
    "1": [
        [
            0.4444444444444444,
            "3"
        ],
        [
            0.2222222222222222,
            "2"
        ],
        [
            0.1111111111111111,
            "0"
        ],
        [
```

- The output:

```
>>> main(2)
[(0.009259259259259259, '1323T'), (0.006172839506172839, '1320T'), (0.006172839506172839, '1233T')]
>>> main(3)
[(0.009259259259259259, '1323T'), (0.008333333333333333, '1303T'), (0.007716049382716049, '1230T')]
>>> main(4)
[(0.009259259259259259, '1323T'), (0.008333333333333333, '1303T'), (0.007716049382716049, '1230T')]
>>>
```

2 Shoulder surfer meets Markov II (Bonus)

(a) *Create a markov model of 4- and 6-digit PINs based on the RockYoudata set using Python. You can either filter them by yourself or use the dataset ofMarkert et al.*

- After some preprocessing of the RockYou dataset and error filtering (like in the last exercise sheet) we just filtered the dataset with a regex. Snipped of the code:

```
#create passwordset P1
def filter_4digits_6digits(df):
     #\A from beginning of the string till \z end: looks for digits with 4-4 chars
    or 6-6 chars
     a = '(?:\A(?=\d{4,4}\Z)|\A(?=\d{6,6}\Z))'
     df = df.loc[df['passwords'].str.contains(a, regex=True)].copy()
     #reset index
     df.reset_index(inplace=True, drop=True)
     return df

def main():
    df = preprocess()
    df = correct_errors(df)
    df = filter_4digits_6digits(df)
    write_to_file(df, output_file)
```

Snippet of the filtered dataset:

```
 frequentcys passwords
0 290729 123456
1 13984 654321
2 13272 111111
3 13028 000000
4 9516 123123
5 7419 666666
6 5377 121212
7 5225 112233
8 5175 555555
9 5055 789456
10 5052 999999
11 4576 159753
12 4486 222222
13 4469 987654
14 4118 123321
15 3649 888888
16 3466 456789
17 3141 123654
18 3100 777777
```

(b) *Implement your above algorithm in Python.*

- First we load the dataset and add the bot and top symbol to every PIN

```
def init_database():
     #read datafile
     df = pd.read_csv(datafile, sep='\t', index_col = 0, dtype={'passwords': str})
     #add top and bottom
     df['passwords'] = bot + df['passwords'] + top
     return df
```

- Then we count the occurrences of every char in the alphabet:

```
#count occurrences of chars in the password dataset
def count_occurrences(df, alphabet):
     occurrences = {}
     #for every char in alphabet (digit)
     for char in alphabet:
          #count the occurrence of that char, for every password in the database (* by frequentcys)
          df['occurrences'] = count_char(df['passwords'] , char) * df['frequentcys']
          #save result to dict
          occurrences[char] = df['occurrences'].sum()
     return occurrences

#count occurrence of a char in a string
def count_char(list, char):
     result = []
     for string in list:
          result.append(string.count(char))
     return result
```

```
occ:  {'⊥': 2299585, '0': 2053913, '1': 2333574, '2': 1869088, '3': 1208374, '4': 1050749, '5': 1160702, '6': 1093210, '7': 770912, '8': 1120929, '9': 1094735, 'T': 2299585}
```

- After that we create the alphabet for the Markov model (the edges) and count the occurrences of these chars (edges) as well:

```
#returns an alphabet with every possible combination of the 2 input alphabets
# if you combine the chars in them
#the if clause will prevent that we get edges which will never occur
#e.g. T0, T1, ... TBot, ... 0Bot, 1Bot...
def markov_alphabet(alp1, alp2):
     return [char1+char2 for char1 in alp1 for char2 in alp2 if char1 != top and char2 != bot]
```

```
alphabet_1:  ['⊥', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'T']

alphabet_2:  ['⊥0', '⊥1', '⊥2', '⊥3', '⊥4', '⊥5', '⊥6', '⊥7', '⊥8', '⊥9', '⊥T', '00', '01', '02', '03', '04', '05', '06', '07', '08', '09', '0T', '10', '11', '12', '13', '14', '15', '16', '17', '18', '19', '1T', '20', '21', '22', '23', '24', '25', '26', '27', '28', '29', '2T', '30', '31', '32', '33', '34', '35', '36', '37', '38', '39', '3T', '40', '41', '42', '43', '44', '45', '46', '47', '48', '49', '4T', '50', '51', '52', '53', '54', '55', '56', '57', '58', '59', '5T', '60', '61', '62', '63', '64', '65', '66', '67', '68', '69', '6T', '70', '71', '72', '73', '74', '75', '76', '77', '78', '79', '7T', '80', '81', '82', '83', '84', '85', '86', '87', '88', '89', '8T', '90', '91', '92', '93', '94', '95', '96', '97', '98', '99', '9T']

occ_edge_set:  {'⊥0': 495145, '⊥1': 901009, '⊥2': 330384, '⊥3': 102202, '⊥4': 69034, '⊥5': 70438, '⊥6': 75697, '⊥7': 70500, '⊥8': 102983, '⊥9': 82193, '⊥T': 0, '00': 200589, '01': 225196, '02': 195564, '03': 166737, '04': 151496, '05': 164627, '06': 164048, '07': 160147, '08': 186072, '09': 166437, '0T': 229842, '10': 314843, '11': 270394, '12': 629001, '13': 112622, '14': 112753, '15': 105421, '16': 84293, '17': 89164, '18': 127976, '19': 206565, '1T': 217209, '20': 200787, '21': 210304, '22': 155452, '23': 456015, '24': 101217, '25': 127018, '26': 87732, '27': 85505, '28': 123614, '29': 104409, '2T': 194204, '30': 147571, '31': 127976, '32': 97921, '33': 62935, '34': 334488, '35': 41146, '36': 44623, '37': 33771, '38': 51771, '39': 46387, '3T': 208526, '40': 96401, '41': 82507, '42': 63799, '43': 53213, '44': 37362, '45': 353907, '46': 35807, '47': 38619, '48': 53536, '49': 43370, '4T': 184253, '50': 101754, '51': 87041, '52': 86141, '53': 49625, '54': 58953, '55': 53782, '56': 351858, '57': 37875, '58': 65032, '59': 58922, '5T': 193496, '60': 92870, '61': 75007, '62': 56561, '63': 38717, '64': 27692, '65': 57512, '66': 58760, '67': 40000, '68': 60099, '69': 66159, '6T': 498650, '70': 91789, '71': 73326, '72': 55537, '73': 31838, '74': 33840, '75': 39432, '76': 37963, '77': 54659, '78': 91420, '79': 61854, '7T': 187976, '80': 116225, '81': 95484, '82': 79236, '83': 56231, '84': 59428, '85': 77501, '86': 74460, '87': 95218, '88': 128198, '89': 132981, '8T': 189670, '90': 152781, '91': 121997, '92': 96661, '93': 66980, '94': 56511, '95': 53695, '96': 56786, '97': 54176, '98': 113931, '99': 108707, '9T': 195759}
```

- And at last we calculate the Markov modell with these occurrences:

```
#creates a makrov model / calcs the probability of the edge_set
def create_makrov_model(occ, alphabet, occ_edge_set):
     markov_model = {}
     for char in alphabet:
          #divides the total num of edges by the total number of the beginning vertice
          #e.g. occ_edge_set['01'] = 10 / occ['0'] = 20  => 0.5
          markov_model[char] = occ_edge_set[char] / occ[char[0]]
     return markov_model
```

(c) *Evaluate your algorithms. What is the advantage of your algorithms compared to a random guess?*

- If we use the Markov model like in task 1. and output three likely PINs when we observe one digit. We get the following results:

```
>>> main(2)
[(0.003192080901017929, '1234T'), (0.0008987235750735024, '1230T'), (0.0008493872205011611, '1212T')]
>>> main(4)
[(0.003192080901017929, '1234T'), (0.0009438533961205413, '1223T'), (0.0013149627524600069, '1123T')]
>>> main(6)
[(0.003192080901017929, '1234T'), (0.0025328192843752464, '1256T'), (0.002250275131115608, '1456T')]
>>> main(8)
[(0.003192080901017929, '1234T'), (0.0025328192843752464, '1256T'), (0.002250275131115608, '1456T')]
>>> main(10)
[(0.003192080901017929, '1234T'), (0.0025328192843752464, '1256T'), (0.002250275131115608, '1456T')]
```

These PINs are better than a random guess, as they are much more likely to occur. A random guess would have the probability of $\frac{1}{10^3} = 0.001$, which is just a third compared to our most likely PIN: 0.0032.

3 Follow up

Please read Sections 4.1 and 4.2 of the paper3and answer the following questions:

(a) *What are the differences to the algorithm from the lecture?*

(b) *Why do you think Algorithm 1 is different?*

(c) *What are the consequences for storage and runtime?*

Compare algorithm from the lecture to algorithms their algorithms:

(a) *partial_size1(current_length, level)*

(b) *partial_size2(current_length, prev_char, level)*

(c) *Is a table size of maxlength · maxlevel enough?*

1. In the lecture l stands for level / rank, while in the note, it stands for the length of a password
2. In the lecture p_1 or probability(e_1) stands for the probability of an char / edge, while in the note they use ν (Ny)
3. For the probability of a path / password, they use the symbol θ (Theta)
4. They don't show how to save the save the information, which is goatherd with partial_size1. While algorithm 1 does show that the information gets stored in a table.
5. The partial_size1 algorithm supposes that the Markov model is of order 1. While the Algorithm 1 does not.
6. The partial_size1 and 2 algorithms only allow passwords with a fixed (same) length, while the algorithm from the lecture allows passwords of different sizes.
7. Also the table size of max_length · max_level is not enough. In the paper for partial_size1 they argue that a 2D-Array of size l times number of levels should be sufficient. That is the case for partial_size1, as each probability is independent of one another. But not for partial_size2, because it works for Markov models order 1, thus it loses the independence of probability and a larger table will be needed.